Tutorial Dhaka Trainings. Session 2. Day 2. DP

October 10, 2022

A. Minimums on the Edges

First, let's precalculate, for each subset of vertices, the number of edges between them. It can be done in $O(2^n \cdot n \cdot n)$. Then, let's solve the problem using dynamic programming. Let's calculate dp[i][j]: the maximum possible sum if we consider vertices in mask i and distribute j tokens between them. So there are only several transitions:

- First, we can delete one node from the mask because we don't want to add more tokens to it
- Second, we can add one token to each node from the mask, and add the number of edges inside it to the answer.

So, the values of the dynamic programming function can be calculated in $O(2^n \cdot s \cdot n)$.

B. Snowy Smile

Let's fix the top of the rectangle, and consider each pirate chests from top to bottom. Now it becomes a dynamic 1D version, which can be speeded up using segment tree. Overall time complexity is $O(n^2 \log n)$.

C. Bitwise Xor

Several solutions exist, one of them is the following: Using bitwise trie it is easy to prove that the minimum of $(a_i \oplus a_j)$ is achieved at some two adjacent numbers.

So you can sort everything, calculate dpi is the number of good subsequences ending at element i, and recalc it as $dp_i += dp_j$ if $(j < i)$ and $a_i \oplus a_j \geq x$.

You can maintain all dp values in the bitwise trie (btw bitwise tree on binary words of length k is equal to segment tree on segment $[0, 2^k)$), to make transitions quickly.

Complexity is $O(kn)$, $k = 60$ in the problem.

D. Make Rounddog Happy

Let's apply Divide and Conquer to this problem and each time we need to calculate the number of legal intervals which cross the middle.

We enumerate the endpoint that the maximum value is closer to, then we can calculate the legal range of the other endpoint, which should satisfy three limits as follow:

- Bigger integers are not allowed. Therefore we can precalculate the first bigger integer on its left/right by a stack;
- Repeated integers are not allowed. Also we can precalculate by a stack;
- The interval's length should be long enough;

The time complexity shall be $O(n \log n)$.

E. Partition Number

Let $p(m)$ be the number of solutions of equation $x_1 + x_2 + ... + x_k = m$ such that $x_1 \leq x_2 \leq ... \leq x_k$. Actually, it is the partition number. And let $odd(x)$ and $even(x)$ be the number of ways summing up to x using odd or even distinct numbers from A, respectively.

The answer for the problem will be:

$$\sum_{i=0}^{m}(even(i) - odd(i)) \cdot p(m-i)$$

which can be proved using inclusion-exclusion principle.

The value of $odd(x)$ and $even(x)$ can be calculated using simple dynamic programming. And the value of $p(x)$ can be calculated using the classic dynamic programming or pentagonal number theorem in $O(m\sqrt{m})$. Also, polynomial inversion and fast Fourier transform work in $O(m \log m)$.

F. Inv

A permutation and its inverse have the same number of inversions. Additionally, a permutation is an involution if and only if it is the same as its inverse. So the parity of the number of involutions with given number of inversions is equal to the parity of the number of all permutations with given number of inversions: all non-involutions can be paired up. And the latter number can be calculated by dynamic programming.

The complexity is $O(n^3)$.

G. Dense Subgraph

The subgraph with the largest density is always a star.

If the tree is not a star, we can divide it to two trees with at least two vertices, and that means that at least one of these trees have density not less than of initial tree. So, there exists $O(n \cdot 2^{deg})$ possible subgraphs that could have the largest density. We need to make sure that all of them have density $\leq x$.

For each such subgraph let's check that it can't be presented in the set of turned on vertices.

In $O(n \cdot 2^{deg} \cdot deg)$ we can find all possible sets of turned on neighbors of each vertex. After that, it is possible to calculate the answer using simple DP on the tree.

H. Gurdurr

Let's denote the types of layers. (□ — nonexistent block, ■ —- existent block.)
$A =$ □■□
$B =$ ■■□ = □■■
$C =$ ■□■
$D =$ ■■■

A straightforward dynamic programming would have $O(4^n)$ states and would make transitions in $O(n)$ time, which is definitely too slow. Let's make some observations.

Firstly, if we have some layer of type C, then we cannot touch it anymore. On the other hand, it can be adjacent to layers of any other type, so we can think about such a layer as a separator between two independent games. Using this fact we can reduce the number of states to $O(3^n)$ if we always split the jenga towers on each layer of the type C and calculate NIMBERS for each possibility instead of only calculating whether some configuration is wining for the current player or not.

This is still too much. Now, let's look at the layers of the type A. If such a layer touches some layer of type B or C, we cannot touch them anymore. We could almost split towers on the layers of the type A, but remembering about the fact, that together with such layer we need to remove the neighboring ones. The problem is that if a layer of type A touches a layer of type D, we still can remove one block in the future. Fortunately, this one removable block is independent from anything other in the game and also it doesn't matter if we change the layer of type D into a layer of type B or C. In other words, this single block becomes an independent game with $NIMBER$ equal to 1, which is easy to be considered. For instance, a stack of layers of types $BBDBADDBD$ is equivalentto a sum of three games: a jenga tower BBD, a jenga tower DBD, and a single independent game with a single move (as D is adjacent to A).

Getting rid of layers of type A and C we can decrease the number of states to $O(2^n)$ and the whole complexity to $O(2^n \cdot n)$, which is sufficient. Also note that after such preprocessing we can easily answer each query in O(n) time.

I. Hypno

Compute the DP for each vertex: $T(v)$ – optimal expected time necessary to reach n from v. Of course, $T(n) = 0$. Consider a single vertex v. And assume we know T's for each neighbor of v, and these T's are sorted: $e_1 \leq e_2 \leq ... \leq e_k$. We can show that:

- If we didn't use an edge at all, it will succeed with $\frac{3}{4}$ probability ($\frac{1}{2}$ chance to be immune to a Hypno on that edge, and $\frac{1}{2} \cdot \frac{1}{2}$ to get lucky with a Hypno we're susceptible to).
- If we used an edge and it fails, we know we're susceptible to the Hypno on it. From now on, the edge will succeed with $\frac{1}{2}$ probability.
- The optimal strategy looks as follows: for some $j \in [1, k]$, try to go to e_1; if unsuccessful, try e_2 etc. If the edge leading to vertex e_j fails, repeatedly try to go to vertex e_1.
- $T(v) \geq e_j + \frac{2}{3}$. Therefore, if we compute the T's for each vertex in the increasing order of times, all neighbors of $vneeded$ for $ourstrategy$ will be computed far ahead of v.

With the knowledge above, we can use a variant of Dijkstra algorithm to compute all $T's$: for each yet undiscovered vertex v, maintain the current candidate for $T(v)$ basing on the neighbors of v discovered so far. When another neighbor (s) of v is discovered, update the candidate. We can use the fact that $T(s)$ is the largest time discovered so far and therefore it can only be the last considered neighbor in the freshest strategy for v.

When implemented correctly, the solution can be implemented in $O(m \log n)$ time.

J. Square Substrings

We call the substring $s_i s_{i+1} ... s_j$ a run, denoted by (i, j, d), if there exists a smallest d such that

- $2d \leq j - i + 1$;
- $s_k = s_{k+d}$ for $k = i, i + 1, ..., j - d$;
- $s_{i-1} \neq s_{i+d-1}$ if s_{i-1} exists;
- $s_{j+1} \neq s_{j-d+1}$ if s_{j+1} exists.

With respect to a strict total order $<$ on the alphabet $\sum$, we call a string w Lyndon word if $w < u$ is true for each proper suffix u of w. Since for a run (i, j, d) we have $s_{j+1} \neq s_{j-d+1}$, we can always find a Lyndon word $s_u s_{u+1} ... s_v$ with respect to a strict total order $<$ or its reversed order, such that $v - u + 1 = d$ and $i < u \leq i + d$. Also, it can be proved that different runs don't share any such words, and thus the number of runs is $O(n)$.

By further proof, we can get $\sum_{(i,j,d) \text{ is a run}} \frac{j-i+1}{d} \leq 3n - 3$, which implies that, if we pick up all the squres as square-triples (L, R, k) such that each length-k substring of $s_L s_{L+1} ... s_R$ is a square, the number of these square-triples are less than $\frac{3}{2}n$. If we can pick up these square-triples, we can solve the problem using some data structures.

First, let's talk about how to get these triples. There are many ways to detect runs, and then you can pick up all the square-triples. For example, you can enumerate d and positions $1, d+1, 2d+1, ...$ in the string so that you can check if two consecutive positions are in the same run and then detect the run. Alternatively, you can linearly build the Lyndon tree of the string and meanwhile get runs. The former detects all runs in time complexity $O(n \log n)$, and the latter in $O(n)$, in which building the suffix array of the string in linear time is required.

Then, let's see if we can maintain some data structure to solve the problem in time complexity $O((n + q) \log n)$. Let's consider the relationship between a query (l_i, r_i) and a square-triple (L, R, k). Obviously, if the triple has any contribution to the query, it must have $k \leq r_i - l_i + 1$. Furthermore, we can split one triple (L, R, k) into two triples without upper bounds, (L, ∞, k) and $(R - k + 2, \infty, k)$, so that we can handle things easily. That is, for a new triple (u, ∞, k), its contribution (without regard to its sign) to the query (l_i, r_i) is $(max\{r_i - u - k + 2, 0\} - max\{l_i - u, 0\})$, which can be calculated by maintaining two Fenwick trees and enumerating intervals in increasing order of length.

K. Eevee

Precompute all binomial coefficients and factorials up to nk, for sure they'll be useful.

Let's solve the problem for all permutations, not for intervals of them. Add a number $n + 1$ at the end of every permutation and assume that we are also interleaving these new numbers, all of them must be at the end of the sequence (so we allow and even force them to be neighboring), and still no other number can have all its occurrences neighboring. This will make implementation easier. At the end we'll have to divide the result by $k!$, as we don't really choose the order of these k new numbers.

For each permutation, let its x-prefix be the prefix ending at x. Let's denote by $DP[x]$ (for $1 \leq x \leq n + 1$) the number of ways to interleave the x-prefixes of all permutations so that the numbers x are neighboring at the end of the resulting sequence

and there are no other numbers which have k neighboring occurrences. It's clear that the answer will be equal to $\frac{DP[n+1]}{k!}$. How to calculate $DP[x]$? The total number of ways to interleave all the x-prefixes so that all the numbers x appear at the end of the resulting sequence (but allowing other numbers to have k neighboring occurrences as well) canbe easily computed — it's a product of some binomial coefficients and factorials. We then have to subtract the ways in which some other number has k neighboring occurrences. Let's iterate over first such occurrence y. This scenario is only possible if y appears before x in all k permutations. The number of ways to create such interleaving is also a product of some binomial coefficients, factorials and $DP[y]$.

So, this is a solution which works in complexity $O(n^2 \cdot k)$, so we already know how to solve thewhole task in $O(n^2 \cdot k^3)$, which is way too slow. The first idea is to calculate the result faster for all prefixes of the permutation sequence (we'll then repeat the computation k times, iterating over the start of an interval). This can be done as the factor k came from the fact that for each pair (y, x) we had to calculate the product of factorials of distances between ys and xs in all permutations in the interval. We can do it faster by maintaining this product and updating it as we add new permutations to the prefix. This gives us a solution in $O(n^2 \cdot k^2)$ time — still too slow.

Now we have to use the randomness of the permutations. The complexity in the original problem wasn't exactly $O(n^2 \cdot k)$, it was
$O(n \cdot k+ (numberof\ pairs\ (y, x)$ such that y in every permutation comes before $x) \cdot k)$, as it's the number of transitions in our dynamic programming. Let's estimate this number. If the prefix contains only one permutation, we have exactly $(\frac{n+1}{2})$ such pairs. Nextly, after adding each new permutation to the prefix, the number of such pairs approximately halves. Therefore, the expected number of such pair over all prefixes is also $O(n^2)$. We can use this fact and implement the solution by maintaining the set of good pairs, and performing only necessary DP transitions. When implemented properly, the solution has complexity $O(n \cdot k \cdot (n + k))$.